UNITED STATES NON-PROVISIONAL PATENT APPLICATION FOR:

METHOD AND SYSTEM FOR CODE OPTIMIZATION

INVENTOR:

MICHAEL BOUCHER

CERTIFICATE OF MAILING BY "EXPRESS MAIL"
UNDER 37 C.F.R. § 1.10

"Express Mail" mailing label number: EV433241380US
Date of Mailing: May 4, 2004

I hereby certify that this correspondence is being deposited with the United States Postal Service, utilizing the "Express Mail Post Office to Addressee" service addressed to Mail Stop PATENT APPLICATION, Commissioner for Patents, PO Box 1450, Alexandria, VA 22313-1450 and mailed on the above Date of Mailing with the above "Express Mail" mailing label number.

METHOD AND SYSTEM FOR CODE OPTIMIZATION

Related Application Informaion

This application is related to United States Patent Application Serial No. ______________ filed concurrently herewith, and entitled "METHOD AND SYSTEM FOR TARGETING PROFILE GATHERING THROUGH REAL-TIME DATA," Attorney Docket No. 30014200-1108/SUN030075, and which is hereby incorporated by reference herein.

Field of the Invention

The present invention generally relates to data processing systems. In particular, methods and systems consistent with the present invention generally relate to optimization of executable code.

Background of the Invention

Computers are increasingly important in today's society, and software used to control computers is typically written in a programming language. C, C++ and other similar variations are widely used programming languages. The programming language C is described, for example, in detail in Al Kelley et al., "A Book on C," Addison-Wesley, 1997, which is incorporated herein by reference. In developing software, typically a software developer writes code, referred to as "source code," in a programming language, and the source code is compiled by a compiler into "object code" that can be run by a machine. This code is also referred to as executable code.

Although it may be easy to generate executable code that works, it may be difficult to generate executable code that operates optimally. Code that may produce correct results may do so in a manner that is inefficient, use too many system resources or take too long to execute properly. The problem of generating optimal executable code from a given source code may be generally difficult. Optimal execution may refer, for example, to obtaining peak speed, highest scalability of a parallel program, shortest time to solution, sharpest intervals in the result, smallest memory footprint, most efficient use of parallel processors, etc. Optimal execution may also refer to other performance areas.

Although the problem of optimization may be generally difficult, executable code generated directly from a specification such as a source code file, intermediate representation, or an object file can often be substantially improved by compilers, binary optimizers, and other tools that attempt to produce executable code. The process of attempting to generate code approaching an optimal solution may be referred to as optimization. Optimization is also made difficult by the absence of information required for a true optimum to be reached. Such information may include, for example, the probability with which a particular decision will be made one way or another, the frequency with which particular data will be used, the size of a particular data set and many other factors that inhibit optimization.

One solution to the problem of insufficient information, referred to as "profile feedback," is to create an executable program that contains "instrumentation" and then recompile with the results of the instrumentation. The instrumentation may be additional code that gathers some of the information useful in optimization. For example, given a decision point for which it is useful to know the probability with which particular decisions are made, one form of instrumentation is counters on all possible decisions. The instrumented program is run, data from the counters is saved, and that data may be used by a recompilation of the program to optimize the code around the decision point.

Profile feedback has numerous substantial drawbacks that prevent its widespread adoption. For example, it may be intrusive and change the character of the program that it is measuring. This means that the result of thorough instrumentation is to gather instrumentation on the instrumented program rather than on the program of interest. Another difficulty is making the choice between high instrumentation, which is intrusive and can substantially slow the program execution, or low instrumentation, wherein necessary information may not be gathered. Profile feedback may also require a second compilation phase which may be expensive on time and system resources.

Another solution for optimization is statistic analysis. In this method, the program is analyzed without benefit of any information except what is expressed in the representation of the program. Algorithms are applied to try to decide various optimization-related questions. The algorithms are often quite expensive and error-prone and often fail to decide the question. Static analysis is an ordinary process that a compiler applies to try to analyze a program. Static analysis is in contrast to dynamic analysis, which is a type of analysis that occurs at run-time and uses information generated by observing the dynamic run-time environment.

By contrast, static analysis has available the information available to the compiler at the time the program is compiled. Consider the following example:

```
      DO 20, I = 1, N
            DO 10, J = 1, M
                  X = A(I) * B(J) / 2.0
10 END DO
20 END DO
```

In this example, the static compiler can determine that it is better to multiply the 0.5 instead of divide by 2.0 because multiplication is faster than division. However, it is best to have the loop with the highest iteration count as the outer loop, and the static compiler cannot guess from analyzing this code segment whether it should accept the code as-is or interchange the nesting order of the loops.

Yet another optimization solution is directives. In this method, the information in the program is augmented with directives conveying information that may be useful in optimization. These directives are typically stored in the source code as comments. The strengths of this system are that it is a simple way of providing optimization-related information to the compilation system. However, directives have several drawbacks. For example, the directives are typically written by a human, which can be a time-consuming and error-prone process. Additionally, the directives typically provide a very small subset of the information required for best optimization. The directives are typically not portable between vendors, compilers, or over time. In practice, this means that the directives are not up-to-date for any particular target environment.

A directive may be information embedded in a comment that tells the compiler something interesting about the program. For example:

```
!$OMP  PARALLEL  DO
         DO  10,  I  =  1,  N
                  CALL SUBR X(I))
   10  END  DO
!$OMP  END  DO  NOWAIT
```

The "!" character indicates a comment, for example, in Fortran. When this code is presented to a compiler that does not understand the directives, it treats them as comments and ignores them. When this code is presented to a compiler that does understand the directives, then it realizes that the first directive indicates that it is allowed to parallelize the loop. The second directive (END DO NOWAIT) indicates that the compiler is allowed to generate code that does not force each parallel thread to wait for the others at the end of the loop. Ordinarily, the threads would finish their work and then wait for the others before proceeding. The NOWAIT tells them to proceed without waiting.

Therefore, a need has long existed for a method and system that overcome these and related problems.

Summary of the Invention

Methods, systems, and articles of manufacture consistent with the present invention optimize a program using a statistical profile and accumulated information about prior compilations and optimizations. When the program is compiled or optimized an optimization record is updated that includes changes made during the compilation or optimization. Therefore, when additional compilations or optimizations are performed, the optimization record can be reviewed along with the statistical profile to determine the results of the changes.

In accordance with methods consistent with the present invention, a method in a data processing system for optimizing a program is provided. The method comprising the steps of automatically analyzing a statistical profile of a program execution, and automatically optimizing the program based on at least one of the analysis, information about at least one prior compilation of the program, and information about at least one prior optimization of the program.

In accordance with articles of manufacture consistent with the present invention, a computer-readable medium containing instructions that cause a data processing system to perform a method for optimizing a program is provided. The method comprises the steps of automatically analyzing a statistical profile of a program execution, and automatically optimizing the program based on at least one of the analysis, information about at least one prior compilation of the program, and information about at least one prior optimization of the program.

In accordance with systems consistent with the present invention, a data processing system for optimizing a program is provided. The data processing system comprises a memory having an optimization program that automatically analyzes a statistical profile of a program execution, and automatically optimizes the program based on at least one of the analysis, information about at least one prior compilation of the program, and information about at least one prior optimization of the program. The data processing system further comprises a processing unit that runs the program.

In accordance with systems consistent with the present invention, a data processing system for optimizing a program is provided. The data processing system comprises means for automatically analyzing a statistical profile of a program execution, and means for automatically optimizing the program based on at least one of the analysis, information about at least one prior compilation of the program, and information about at least one prior optimization of the program.

Other features of the invention will become apparent to one with skill in the art upon examination of the following figures and detailed description. It is intended that all such additional systems, methods, features, and advantages be included within this description, be within the scope of the invention, and be protected by the accompanying drawings.

Brief Description of the Drawings

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate embodiments in accordance with methods and systems consistent with the present invention and, together with the description, serve to explain the advantages and principles consistent with the present invention.

Figure 1 depicts an exemplary data processing system suitable for use in accordance with methods and systems consistent with the present invention.

Figure 2 depicts steps in the method for gathering a statistical profile in accordance with methods and systems consistent with the present invention.

Figure 3 depicts steps in an exemplary method for optimizing a program in accordance with methods and systems consistent with the present invention.

Detailed Description

Reference will now be made in detail to an implementation in accordance with methods, systems, and articles of manufacture consistent with the present invention. Wherever possible, the same reference numbers will be used throughout the drawings and the following description to refer to the same or like parts.

Methods, systems, and articles of manufacture consistent with the present invention optimize a program using a statistical profile and accumulated information about prior compilations and optimizations. When the program is compiled or optimized an optimization record is updated that includes changes made during the compilation or optimization. Therefore, when additional compilations or optimizations are performed, the optimization record can be reviewed along with the statistical profile to determine the results of the changes.

The statistical profile includes execution characteristics of the program, which can be used to regenerate the program's code at run-time. By relying on the statistical profile and the optimization record of the program, methods, systems, and articles of manufacture consistent with the present invention minimize the problems of instrumented code. By generating and regenerating code at run-time, methods and systems in accordance with the present invention avoid the problems of a separate recompilation phase. By working at run-time, methods and systems in accordance with the present invention minimize the problems of incomplete information.

When the program is compiled for the first time or subsequent times or when the program is optimized, an optimization record is updated that includes a record of changes made to the program. The optimization record includes, for example, programming structures implemented in the program, such as loops and function. Further, the optimization record includes changes that are made to the program during optimization or recompilation, such as inlining a function, unrolling a loop, cloning a function to create a different version of the function, and performing a speculative execution. One having skill in the art will appreciate that the optimization record is not limited to the items described above and can include additional information. Further, the optimization record can be included in the statistical profile data. The optimization record is generated and updated by the profiler, which receives information about program updates made during compilation from the compiler.

To gather a statistical profile, methods and systems in accordance with the present invention cause the system to generate periodic or non-periodic interrupts throughout the execution of the program. The interrupts delivered to a particular thread of execution may or may not be correlated with the interrupts delivered to any other thread. For example, to get a view of the execution pattern of the entire system, a system could deliver periodic interrupts to every thread. However, it may be useful to try to understand the state of a subset of the system. For example, the run-time compiler may create diagnostic code to see whether a loop iteration count or cache residency assumptions are violated. It may not be desirable to keep the diagnostic code there for long because the code may become a drag on performance with frequent checks and measurements. In this case, the run-time compiler may set an interrupt for those threads that are using the instrumented code to remind itself to check the results of its diagnostics. This interrupt does not apply to the entire system and may be received by a single thread. Its arrival is not correlated with signals arriving at other threads or for other reasons.

On each interrupt, the system records a snapshot of the execution state of the system. The state may be characterized by a call chain for one or more of the threads of execution, address of the currently executing instruction, address of data being read and other characteristics. The call chain at a particular point defines how execution is reached that point. For example, if function F1 calls function F2, which then calls function F3, then the call chain would be F3->F2->F1. All or part of the snapshot may be stored for later use, and the snapshots are recorded for later analysis.

At some point during execution, a thread of execution analyzes the statistical profile and possibly additional data and uses it to attempt to further optimize the program while it is running. The additional data may include information left by the compiler that is useful to the optimizer. For example, consider the following code:

```
     DO 10, I = 1, N
            CALL SUB1 (I)
  10 END DO
```

If the code generator initially assumes that this loop accounts for a small amount of execution time, then it may not inline SUB1. If dynamic analysis shows that it accounts for more execution time, it may then consider inlining SUB1. However, some subroutines are not candidates for inlining and the code generator may have to spend time at run-time determining whether SUB1 is a candidate. If the compiler has left information indicating that SUB1 can or cannot be inlined, then the code generator can save the time of computing that at run-time. This is beneficial because time spent at run-time is time included in the cost of running the program whereas time spent in compilation is relatively less expensive.

The system determines areas in which additional optimization may be applied profitably. One exemplary criterion for determining that optimization may be applied profitably is determining that a particular part of the program consumes a substantial portion of the run-time. Another exemplary criterion for determining that optimization may be applied profitably is observing that many processors are idle waiting for the results of a particular computation. Another exemplary criterion for determining that optimization may be applied profitably is determining that a particular computation runs more slowly than predicated with a model.

Once the system identifies a section of the program to be optimized, it attempts to apply additional optimization. In doing so, the system analyzes the optimization record to determine what changes may have already been made to that section. For example, if the system previously attempted to optimize the section using a first approach, which did not produce adequate results, then the system may attempt using a different approach. If it is unable to decide a question relating to optimization, it recompiles that section of code with instrumentation to help it answer that particular question. It then allows the program to continue running long enough to gather sufficient data to answer the question. When it has enough data to answer the question, it again attempts to compile the code as described above. It will either discover that it needs additional information, in which case it will iterate on the process described above, or it will have what it needs and will compile a new version of the code.

The additional information previously mentioned used in addition to the statistical profile might tell the code generator what questions are most likely to yield answers that help the performance problem. The additional information may be supplied by the compiler. For example, consider the following loop:

```
DO 10, I = 1, N
SUM = SUM + X(I)
10 END DO
```

The compiler may note that it assumes that the iteration count will be high and that the X array will be resident in cache. Then if the loop runs poorly and the code generator wants to know why, it can consult the additional information, in this case the assumptions that the compiler made, to try to figure out what went wrong. In this particular case, it could generate diagnostic code to verify that the loop iteration count is actually high and that the code does not have excessive stalls loading X values. It could then come back in a few seconds to see what its diagnostic code had found. If it found that iteration counts are high but stalls waiting for X are also high, it could regenerate the loop with code that assumes that X is not cache resident. In that case, it might generate code that contains prefetch instructions to fetch data from memory in advance of when it will be used.

Figure 1 depicts an exemplary data processing system suitable for use in accordance with methods and systems consistent with the present invention. Figure 1 shows a computer 102 which may be connected to a network, which may be wired or wireless, and may be a LAN or WAN, and the computer may represent any kind of data processing device, such as a general-purpose data processing device, a personal computer, a plurality of interconnected data processing devices, video game console, clustered server, a mobile computing device, a personal data organizer, a mobile communication device including mobile telephone or similar devices. The computer 102 may represent a computer in a distributed environment, such as on the Internet. There may also be more computers 102 than shown on the figure.

A computer 102 may include a central processing unit ("CPU") 104, an input-output ("I/O") unit 106 such as a mouse or keyboard, or a graphical input device such as a writing tablet, and a memory 108 such as a random access memory ("RAM") or other dynamic storage device for storing information and instructions to be executed by the CPU. The computer 102 also includes a secondary storage device 116 such as a magnetic disk or optical disk that may communicate with each other via a bus 100 or other communication mechanism. The computer 102 may also include a display 112 such as a cathode ray tube ("CRT") or LCD monitor, and an audio/video input (not shown) such as a webcam and/or microphone.

Although aspects of methods and systems consistent with the present invention are described as being stored in memory 108, one having skill in the art will appreciate all or part of methods and systems consistent with the present invention may be stored on or read from other computer-readable media, such as secondary storage devices, like hard disks, floppy disks, and CD-ROM; a carrier wave received from a network such as the Internet; or other forms of ROM or RAM either currently known or later developed. Further, although specific components of the data processing system are described, one skilled in the art will appreciate that a data processing system suitable for use with methods, systems, and articles of manufacture consistent with the present invention may contain additional or different components. The computer 102 may include a human user or may include a user agent. The term "user" may refer to a human user, software, hardware or any other entity using the system.

As shown, the memory 108 in the computer 102 may include a program instance 117 having user code 118 and a static compiler 114 which compiles code from a human-readable form (*e.g.*, C, Fortran) into a machine-readable form. This form may be object code or executable code suitable for execution, byte code suitable for interpretation (*e.g.*, Java), or an abstract representation intended for later transformation by other components in the system (*e.g.*, the run-time compiler described below).

The run-time compiler 122 generates executable code based on the results of static compilation, analysis of various execution profiles, analysis of the optimization record 126, and consideration of environmental details such as number of CPUs, system load, and amount of available memory. The run-time compiler 122 takes the output from the static compiler 114 plus possibly other information such as the profile data and may recompile all, some, or none of the code. When the run-time compiler makes changes to the program during compilation or during optimization, the run-time compiler updates the optimization record with information about the changes. In one implementation, the run-time compiler 122 compiles all of the code if the static compiler 114 produces only an intermediate form that cannot be executed. This may happen, for example, in delayed binding systems in which the executable file is expected to run on many different systems such as an Intel x86 and a Sun SPARC. In this case, the instruction generation cannot take place until the executable sees the target machines. The run-time compiler 122 may compile some of the code if the static compiler 114 produces code that does not run fast. In one implementation, the run-time compiler 122 may compile none of the code if the static compiler 114 produces code that runs fast enough. For example, if a particular run of a program has a short run-time, then the run-time compiler 122 may not be invoked during that particular run.

The run-time compiler 122 may detect a problem at run-time and then generate code to determine what the problem is. For example, consider the following exemplary code:

```
DO 10, I = 1, N
        X(I) = 526 * X (I)
10 END DO
```

The run-time compiler 122 may make various assumptions about the loop and then generate code based on those. For example, if it assumes that the iteration count (N) is very large, then it may create code that takes a while to start up but that runs very fast once it gets going. Techniques used in such a loop may include prefetching data from memory and unrolling the loop.

If it turns out that the loop iteration count is actually very small, *e.g.*, 3, then the prefetching and unrolling will not help, and the overhead of setting them up will make the loop run more slowly. If the run-time compiler 122 notices the loop above running slowly, it may consider two possibilities: (1) the unrolling or prefetching are not being done well, and (2) the iteration count is much smaller than it assumed and it should not unroll or prefetch.

In response, the run-time compiler 122 could generate exemplary code such as the following:

```
        NITER = NITER + N
        NLOOP = NLOOP + 1
```

```
      STALL_CYCLES  = GET _STALL_CYCLES ( )
      DO  10,  I  =  1,  N
            X (I)  =  526  *  X (I)
   10 END  DO
      TOTAL_STALL  =
   $  TOTAL_STALL  +  (GET_STALL_CYCLES ( )  -  START_STALL_CYCLES)
```

Then it could let the code run for a while longer and then perform further analysis. As a result, it may apply the following considerations. If the average iteration count computed with NITER / NLOOP is smaller than an expected value or threshold then that indicates that the loop should be compiled without the high-overhead techniques that improve long loops and should be recompiled with low-overhead techniques that help short loops. If the amount of time stalled on memory indicated by TOTAL_STALL is higher than an expected value or threshold then that indicates that the loop has a higher-than-expected dependence on the time to fetch operands from memory. In this case, it may recompile the loop with more aggressive prefetching if possible, or otherwise leave it alone. If neither are applicable then the source of the problem may be unknown, in which case it may remove the instrumentation code and attempt a different compilation strategy.

The profiler 120 comprises a data collector 126 and an analyzer 128. The data collector 126 collects statistical and event-driven run-time information and stores it as statistical profile data 124. It collects the event-driven information by establishing instrumentation points to count the number of times that certain events occur and/or how much time they take. For example, it may set such a point in a subroutine that attempts to lock a mutual exclusion lock and then count how many times the lock was already locked by another entity. It may also record how much time it spent waiting for the lock when it was held by another entity. The data collector 126 may collect statistical run-time information by establishing counters that generate interrupts when they count down to zero or when they overflow. A common example of this is a count-down timer but other examples include counters that count the number of cycles spent waiting for an event such as satisfaction of a memory request. When the interrupt is generated, a part of the data collector 126 catches the interrupt indication and takes a snapshot of part of the system state. The state is then recorded, the counter is reset, and control is returned to the code that was interrupted.

Periodically, or in response to specific stimli such as a function call, an analyzer 128 runs that analyzes the statistical profile data 124 and determines where the points of interest are and what can be done. It invokes various systems components such as the run-time compiler 122 to try to eliminate or mitigate the points of interest where possible. During optimization, the run-time compiler also looks to the optimization record to determine what changes have been previously attempted and the results of the attempts.

Although only one compiler 114 is shown, any number of compilers 114 or other components may be used. Additionally, although shown on the computer 102 in the memory 108, these components may reside elsewhere, such as in the secondary storage 116, or on another computer, such as another computer 102. Furthermore, these components may be hardware or software whereas embodiments in accordance with the present invention are not limited to any specific combination of hardware and/or software. The system may be implemented in any way, in software or hardware or a combination thereof, and may be distributed among many computers. It may represented by any number of components, processes, threads, etc.

Computers 102 may communicate directly or over networks, and may communicate via wired and/or wireless connections, including peer-to-peer wireless networks, or any other method of communication. Communication may be done through any communication protocol, including known and yet to be developed communication protocols. The network may comprise many more computers 102 and than those shown on the figure, and the computers may also have additional or different components than those shown.

It will be appreciated that various modifications to detail may be made to the embodiments as described herein. Further, it is noted that a computer-readable medium may be provided having a program embodied thereon, where the program is to make a computer or system of data processing devices execute functions or operations of the features and elements of the above described examples. A computer-readable medium may include a magnetic or optical or other tangible medium on which a program is embodied, but can also be a signal, (*e.g.*, analog or digital), electromagnetic or optical, in which the program is embodied for transmission. Further, a computer program product may be provided comprising the computer-readable medium.

Figure 2 depicts steps in the method for gathering a statistical profile in accordance with methods and systems consistent with the present invention. In one implementation, these steps are performed automatically at run-time. First, the program or user code 118 to be analyzed is run (step 202) and then statistics are gathered (step 204) by the data collector 126 by generating interrupts throughout the program (step 206). At each of the interrupts in the program, a snapshot of the execution state of the program is taken (step 208) and stored in the statistical profile data base 124 for later analysis (step 210). The analyzer 128 analyzes these snapshots (step 212), and through this analysis attempts to identify part of the program to optimize. If there is no part of the program to optimize (step 214), the program needs no further manipulation.

Figure 3 depicts steps in an exemplary method for optimizing a program in accordance with methods and systems consistent with the present invention. In one implementation, these steps are performed automatically at run-time. The run-time compiler 122 attempts to optimize the program (step 302). It determines if it can decide possibly important questions related to optimization, and if it is unable to decide a question related to optimization (step 304), the run-time compiler 122 recompiles the particular section of code in question with instrumentation (step 306). In one implementation, if the analyzer 128 is able to decide questions related to optimization, the system proceeds to optimize the program (step 312). If there is an undecided question related to optimization and the system has recompiled the particular section of code with instrumentation, it may then run the program long enough to gather sufficient data to decide the question related to optimization (steps 308 and 310). (In one implementation, steps 202-212 may be reiterated for this process). If enough data is gathered (step 310), the run-time compiler 122 may then optimize the program (step 312) and compile the code (step 314). If enough data was not gathered, the system may reiterate the process by having the run-time compiler 122 recompile the particular section of code with further instrumentation and/or running the program long enough to gather sufficient data (step 304).

During compilation, any standard compiler techniques may be used including, but not limited to: (1) optimizing branch generation (using the statistical data to infer branch probability instead of requiring user directives or exact branch counts as gathered by instrumented code), (2) creation of clones based on the cases that arise in frequent call chains, and (3) applying inlining and outlining using the statistical data to infer frequency information instead of requiring user directives or exact frequency information from instrumented code.

The run-time compiler analyzes the optimization record to determine what compiler techniques have already been used on the program. For example, the optimization record may indicate that inlining has already been applied to the section of the program to be optimized.

The following example shows the operation of an exemplary implementation in accordance with methods and systems consistent with the present invention.

```
subroutine sub1(n, ab)
integer n
real ab(n)
do 10, i = 1, n
ab(i) = sqrt(ab(i))
10 continue
end subroutine sub1
```

In this case, the compiler cannot decide whether to use a prefetch instruction to start the process of fetching ab from memory before it is needed. If ab is a large array and it is not in cache, then the prefetch will be profitable. If ab is in the cache, then the prefetch will cause performance degradation by increasing the number of memory instructions issued without getting any corresponding benefit. To decide the question, it recompiles with instrumentation that measures cache misses and lets the instrumented code run for a while.

After enough data has accumulated, it analyzes the data to decide the prefetch question. If references to ab usually miss cache, then it will recompile with a prefetch. If they usually do not miss cache, then it will recompile without a prefetch. If they sometimes do and sometimes do not, then it may use a best judgment or it may recompile with more instrumentation to attempt to figure out the cases in which cache misses occur. For example, it may examine the call chains to see if certain calls tend to result in cache misses whereas others do not.

As an example of a question that may not be decided by the compiler, suppose there is the following code:

```
READ *, N
DO 10, I = 1, N
```

The compiler may not be able to decide if the loop iteration count on the 10 loop will be high or low. In particular, it may not be able to decide if it will be high enough to justify unrolling or parallelization.

In the following example, the compiler may not be able to decide how likely is it that the following branch will be taken and the assignment executed. If it is relatively high, then 1/Q should probably be precomputed. If it is relatively low, then 1/Q should not be precomputed:

```
IF (A .LT. B)  THEN
        Q = 1 / Q
END IF
SUM = SUM + Q
```

In the following example, if the compiler does not have the source to SUB1 at the time that it compiles it, then it does not know whether SUB1 manipulates MYFILE.

```
OPEN (526, FILE="MYFILE")
READ  (526, *)  AB
CALL   SUB1
READ  (526, *)  CD
```

If SUB1 does not affect MYFILE, then the compiler could merge the reads from AB and CD, which can often save time. However, if SUB1 does affect MYFILE, such as reading a value from it, then moving the READ into CD above the call to SUB1 can give wrong answers.

In the following example, if N is large then AB will also be large and operations on AB should assume that it will not fit in cache. If N is small, AB usually does fit into cache and code should be generated accordingly:

```
READ *,  N
ALLOCATED (X (N))
```

An example in which the run-time compiler looks to the optimization record when determining what optimization to make on a section of the program is described below with reference to the illustrative example code section below.

```
      subroutine sub1(m, n, ab, cd)
      integer m, n
      real ab(m,n), cd(n,m)
      do 20, i = 2, m
            do 10, j = 2, n
                  ab(i,j) = func1(i, j, m, n, cd)
   10 end do
   20 end do
      end subroutine sub1


      real function func1(i, j, m, n, cd)
      integer i, j, m, n
```

```
    real cd(m,n)
    if (i .eq. 1) then
          func1 = 0
          do 30, k = 1, m
                func1 = func1 + abs(cd(j,k))
 30 end do
    else
          func1 = sqrt(cd(j,i) + cd(j,i-1))
    end do
    end function func1
```

When the run-time compiler considers the optimizations of inlining function "func1" and unrolling the 10 loop, it is clear that at most one of those is sensible. Using one of those optimizations makes the other work badly. In a case in which the run-time compiler decides to unroll the 10 loop at run-time, the run-time compiler compiles the code with 10 unrolled and continues to run. Later the run-time compiler consults the optimization record and realizes that it has made things worse by unrolling. The run-time compiler again tries to compile the code and again must decide between unrolling 10 and inlining "func1." Using the information that the run-time compiler already tried unrolling, it instead inlines "func1." This gives it a chance to eliminate the if statement in the inlined version of "func1" because the variable i will not be 1.

The run-time compiler has other decisions to make in optimizing "sub1" but may lack the information that it needs to make the decisions well. For example, inlining "func1" may be beneficial if m is small. Switching the order of the "do i" and "do j" loops would optimize the memory access pattern of ab at the expense of cd, but it is hard to tell at compile time which is more important. Loop unrolling is a possibility in at least two of the loops, although one would need to know the iteration count to know for sure and to know the unroll factor. Further, if the run-time compiler does not inline "func1" then it may clone "func1" and create a version that omits the check for i being 1.

The run-time compiler looks to the optimization record and the execution profile to improve its decisions when performing the optimizations. For example, the run-time compiler can look through a list of known optimization techniques that have not already been attempted on the section of the program and choose a technique that is identified as a technique to attempt if the prior technique was unsuccessful. Thus, the run-time compiler can identify logical relations between the known optimization techniques and take the relations into consideration when choosing a technique.

The foregoing description of an implementation of the invention has been presented for purposes of illustration and description. It is not exhaustive and does not limit the invention to the precise form disclosed. Modifications and variations are possible in light of the above teachings or may be acquired from practice. For example, the described implementation includes software but the present invention may be implemented as a combination of hardware and software or in hardware alone. Note also that the implementation may vary between systems. The invention may be implemented with both object-oriented and non-object-oriented programming systems.